Drawings

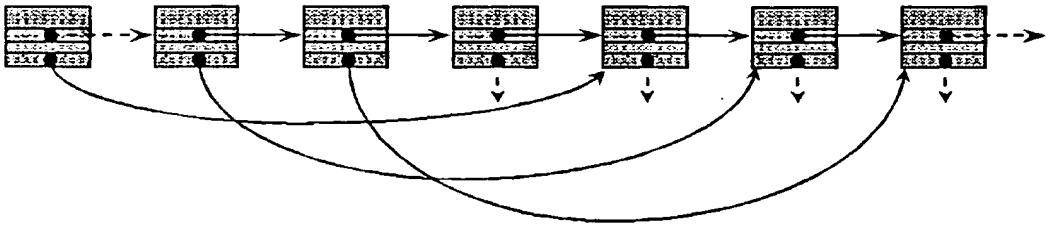

Figure 1: Linked list representation with jump pointers (Prior Art).

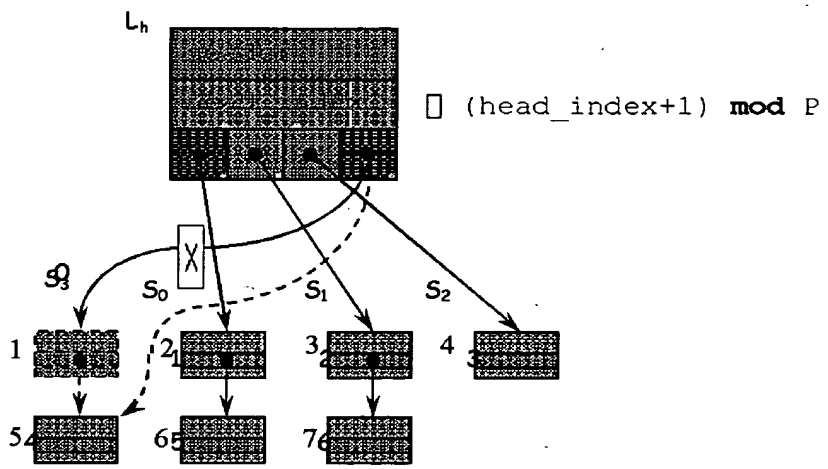

Figure 2: A prefetchable linked list representation.

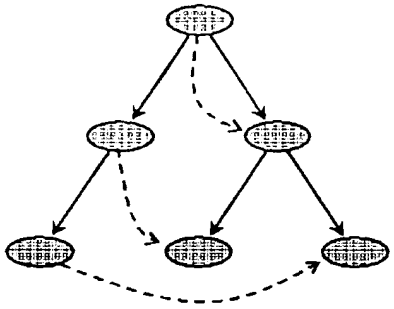

Figure 3: A tree data structure with history pointers (Prior Art).

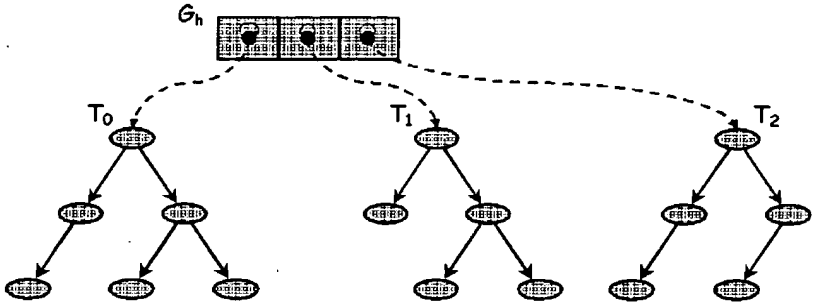

Figure 4: A prefetchable tree representation.

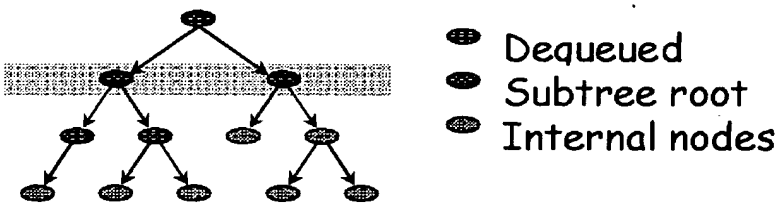

Figure 5: Transforming a Tree into a Forest.

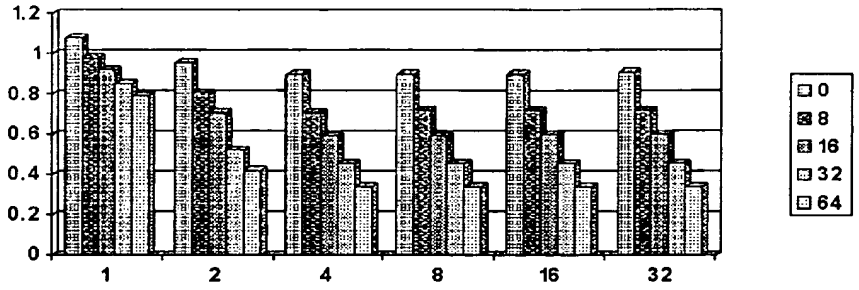

Figure 6: Performance of prefetched linked list traversals.

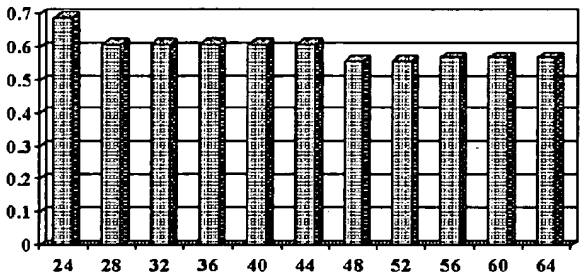

Figure 7: Performance of prefetched tree traversals.

```
list_element_ptr  process_list( list_ptr list )
  {
    int i, p;
    list_element_ptr s[PipeDepth];

    /* prologue */
    p = list->headers;
    for ( i=0, i<p; i++ ) {
      PREFETCH( s[i] = list->head[i] );
    }

    /* steady state */
    while ( p ) {
      for ( i=0; i<p; i++ ) {
        if ( process_element( s[i] ) == STOP )
          return s[i];
        s[i] = s[i]->next;
        PREFETCH( s[i] );
      }
    }
  }
```

Figure 8: Example of a Pipelined Linked List traversal.

```
Traverse( forest_ptr forest )
  {
    /* local variables */
    stack stacks[PipeDepth];  /* PipeDepth stacks */
    tree_ptr n;
    int i, trees_left = PipeDepth;
    struct {
      tree_ptr node;
      stack_ptr stack;
    } traversal[PipeDepth];     /* traversal state descriptor */

    /* prologue */
    for ( i=0; i<PipeDepth; i++ ) {
      traversal[i].node = forest->root[i];
      traversal[i].stack = &stack[i];
      PREFETCH(forest->root[i], sizeof(forest->root[i]));
    }

    /* steady state */
    while ( trees_left ) {
      for ( i=0; i<trees_left; i++ ) {
        if ( traversal[i].node->left ) {
          traversal[i].stack->push( traversal[i].node->left );
          traversal[i].node = traversal[i].node->left;
        } else {
          n = traversal[i].stack->pop();
          if ( n == NULL ) {    /* done with tree i */
            trees_left--;
            if ( i != trees_left )
              SWAP( &traversal[i], &traversal[trees_left] );
          }
          process( n );
          traversal[i].node = n->right;
        }
        PREFETCH( traversal[i].node );
      }
    }
  }
```

Figure 9: Example of a Pipelined Tree Traversal.

```
Traverse( tree_ptr tree )
  {
    /* local variables */
        . . .

    /* level-order traversal prologue */
    PREFETCH( tree->root );
    enqueue( src_queue, tree->root );
    for ( i=0, accumulating=true; accumulating; i++ ) {
      n = dequeue(src_queue);
      if ( n == NULL )
        return;              /* we're done */
      process(n->data);

      if ( n->left != NULL ) {
        PREFETCH( n->left );
        enqueue( dst_queue, n->left );
      }
      if ( n->right != NULL ) {
        PREFETCH( n->right );
        enqueue( dst_queue, n->right );
      }
      if ( src_queue->size + dst_queue->size < PipeDepth ) {
        if ( i >= src_queue->size )
          SWAP( src_queue, dst_queue );
      } else {
        accumulating = false;
        while ( src_queue->size > 0 ) {
          traversal[trees_left].node = dequeue( src_queue );
          traversal[trees_left].stack = stack[trees_left];
          trees_left++;
        }
        while ( dst_queue->size > 0 ) {
          traversal[trees_left].node = dequeue( dst_queue );
          traversal[trees_left].stack = stack[trees_left];
          trees_left++;
        }
      }
    }

    /* steady state loop */
        . . .
```

Figure 10: Example of a pipelined level-order tree traversal.